Jedule: A Tool for Visualizing Schedules of Parallel Applications

Sascha Hunold, Ralf Hoffmann, Frédéric Suter

▶ **To cite this version:**

Sascha Hunold, Ralf Hoffmann, Frédéric Suter. Jedule: A Tool for Visualizing Schedules of Parallel Applications. 1st International Workshop on Parallel Software Tools and Tool Infrastructures (PSTI 2010), Sep 2010, San Diego, United States. pp.169-178, 10.1109/ICPPW.2010.34 . hal-00533962

HAL Id: hal-00533962
https://hal.science/hal-00533962

Submitted on 8 Nov 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Jedule: A Tool for Visualizing Schedules of Parallel Applications

Sascha Hunold*
International Computer Science Institute
Berkeley, CA
sascha@icsi.berkeley.edu

Ralf Hoffmann
Department of Computer Science
University of Bayreuth
ralf.hoffmann@uni-bayreuth.de

Frédéric Suter‡
IN2P3 Computing Center, CNRS/IN2P3,
Lyon-Villeurbanne, France
Frederic.Suter@cc.in2p3.fr

***Abstract*—Task scheduling is one of the most prominent problems in the era of parallel computing. We find scheduling algorithms in every domain of computer science, e.g., mapping multiprocessor tasks to clusters, mapping jobs to grid resources, or mapping fine-grained tasks to cores of multicore processors. Many tools exist that help understand or debug an application by presenting visual representations of a certain program run, e.g., visualizations of MPI traces. However, often developers want to get a global and abstract view of their schedules first. In this paper we introduce Jedule, a tool dedicated to visualize schedules of parallel applications. We demonstrate the effectiveness of Jedule by showing how it helped analyzing problems in several case studies.**

I. Introduction

Scheduling limited resources among requesting entities is one of the most challenging problems in computer science. Traditional domains of scheduling algorithms were process or I/O scheduling in operating systems. The current era of computing has undergone a tremendous change of application and hardware design. The number of cores per physical processor has been increasing over the last couple of years and is still growing. On the other hand, fast interconnects made it possible to create large computational grids over the Internet. Each new computational layer from instruction level to host level raises new scheduling problems. Many research projects exist that target the optimization of scheduling algorithms in one computational layer. On grid level, there are job schedulers like Condor [1] that try to optimize the throughput of jobs on a peer to peer grid. On a single subnet, job scheduling of multiprocessor tasks is often found on cluster front-ends. In this case, jobs are defined by a job description file in which the user can reserve resources (usually processors) for an amount of time. The scheduler on the front-end has to decide the order of jobs on the cluster while fulfilling certain criteria, e.g., quality of service. Examples of such schedulers are PBS [2] and Maui [3]. On lower software levels, schedulers assign fine-grained tasks to threads, e.g., tasks schedulers of Intel's Threading Building Blocks [4].

Each scheduling algorithm tries to optimize an objective function. Usually schedulers try to minimize the overall time of a schedule, which often corresponds to maximizing the throughput of the system. In many articles, scheduling algorithms are explained by showing abstract graphics of the key idea but are evaluated in an experiment, measuring, for example, the makespan of a schedule. It is hardly possible for humans to get a rough idea of the entire schedule by only looking at the log files. A new algorithm might perform better than all its competitors in most cases. Nevertheless, there might be corner cases, which could be easily spotted in a graphical representation of schedules. On the other hand, debugging scheduling algorithms is harder without having a graphical representation. A visualization of a schedule lets the user easily do some sanity checks, e.g., checking the number of requested and assigned processors for a multiprocessor job.

Even though scheduling is an important problem in computer science, only a few tools exist that help scientists to develop scheduling algorithms. Most tools are tied to a specific use case like displaying the trace of an execution of a single parallel program [5].

In the present article we introduce the software tool Jedule, which can visualize arbitrary schedules. Originally, Jedule was designed to help develop scheduling algorithms for multiprocessor tasks on clusters and multi-clusters. Over time it has been extended to support all sorts of task schedules.

The remainder of the article is structured as follows. The tool Jedule and its features are described in Section II. In Sections III - VII we describe several scenarios in which Jedule has been successfully applied. In Section VIII we summarize related work and we draw conclusions in Section IX.

II. An Overview of Jedule

A. Properties of a Task Schedule

Schedules are often visualized using Gantt charts, which show the resource utilization over time. Hence, in these two-dimensional Gantt charts, one dimension typically corresponds to the resources of the system (e.g., processors, cores, hosts) and the other dimension corresponds to the time. The utilization of a resource for a limited amount of time is visualized by a rectangle. To depict the utilization of a single resource, the resource axis is equally divided into p segments, where p denotes the number of resources.

In the present paper we focus on task schedules in parallel systems. Many of these tasks are multiprocessor programs or

* This work was supported by a fellowship within the postdoctoral program of the German Academic Exchange Service (DAAD).

‡ This work was partially supported by the ANR project USS SimGrid 08-ANR-SEGI-022.

jobs, e.g., programs written on top of MPI (Message Passing Interface) for distributed memory machines. On the other hand, on a multicore machine, a task can be executed by multiple threads. In both cases, a rectangle in the Gantt chart spans multiple resources. Additionally, a task may require multiple rectangles, when the resources allocated to a task are not contiguous.

B. Requirements for Displaying Schedules

The design of Jedule was driven by the need to support the development of scheduling algorithms on parallel systems. A major requirement is that the tool should be able to support multiprocessor tasks and multi-clusters. Since the viewer targets multitask systems, it is important that Jedule can handle concurrently running tasks of different types, e.g., the overlapping of communication and computation time on a specific host. In these scenarios the schedule viewer should support user defined color maps. Thus, a user should be able to define a color for each type of task.

The schedule viewer should also provide two different modes: an interactive mode and a command line mode. An interactive mode helps the user understand the specific properties of a schedule. Hence, the interactive mode should allow the selection of certain resources (e.g., a cluster) and also enable the user to specify a time frame that he might be interested in (zooming). In the interactive mode the user should be able to retrieve meta information about a task, e.g., showing the list of allocated processors or the node name by clicking on a task in the schedule view.

Additionally, Jedule should also provide a command line interface, which enables the user to leverage Jedule in batch processing. Often the developer of a scheduling algorithm runs many experiments producing hundreds or thousands of schedules. So, Jedule could be used in a pipeline of batch tasks to create schedule graphics for each experiment. In order to provide a powerful command line interface, Jedule should support different output formats like PNG, JPEG, or PDF. Another important requirement is that it should support different style files for drawing a schedule. The style file defines properties of graphic primitives, e.g., font sizes and colors. Supporting external style files makes it easier to tailor schedule graphics for a specific use case. A user might only be interested in a certain task type, so he may highlight this task type by assigning a different color to this type.

C. Jedule in Detail

Schedules are often displayed by simple Gantt charts. Several Gantt chart tools exist, but we needed one that is dedicated to the development of scheduling algorithms. Jedule has been developed to help us understand and tune different algorithms for scheduling multiprocessor tasks.

For portability reasons Jedule was written in Java. Jedule supports all the requirements that were discussed above, e.g., the support of multiple rectangles for a single task to show the resource allocation of a multiprocessor task if the layout of this task's resources is not contiguous.

```
<node_statistics>
  <node_property name="id" value="1"/>
  <node_property name="type" value="computation"/>
  <node_property name="start_time" value="0.000"/>
  <node_property name="end_time" value="0.310"/>
  <configuration>
    <conf_property name="cluster_id" value="0"/>
    <conf_property name="host_nb" value="8"/>
    <host_lists>
      <hosts start="0" nb="8"/>
    </host_lists>
  </configuration>
</node_statistics>
```

Figure 1. XML definition of a task in Jedule (a node can have multiple configurations, e.g., a communication between clusters).

1) Input format: The input file format of Jedule is a custom XML structure. Jedule is bundled with a parser for the current default XML input format. One can also extend Jedule with a different parser and it is therefore possible to have different input formats, not necessarily in XML. However, the basic structure of a schedule that can be displayed by Jedule is always the same, which is defined by the Jedule Java API. On the lowest level, a schedule S consists of v tasks, where each task v_i has a start time t_s and a finish time t_f. A task in Jedule has unique identifier and a type. The type can be arbitrarily chosen by the user and is usually used to group certain tasks together, e.g., computation, communication, or I/O tasks. Since each task can allocate $p_v \leq p$ resources of the system, a task is also characterized by a list of resources R_v with $R_v \subseteq P$ (P is the set of all resources of the system). The developer of scheduling algorithms often wants to group resources together. One example is a multi-cluster, i.e., a system that consists of multiple smaller clusters. These logical clusters might be a commodity cluster running MPI programs or a set of multicore machines. In order to support this notion of multi-clusters, each task in the Jedule format has a reference identifier that defines the corresponding cluster. It is also possible that a task belongs to more than one cluster, so tasks may span different clusters. This is useful if a communication task transfers data between tasks on different clusters or if allocated resources (possibly in a cloud) are scattered across clusters. The clusters $C_j, j \geq 1$, have to be defined in the header of the Jedule input file, and at least one cluster is required. A cluster C_j is also a subset of the overall resources P with $\bigcup_j C_j = P$. The resources between pairs of clusters are disjoint: $C_i \cap C_j = \emptyset$. A sample definition of a multiprocessor task in the Jedule input format is shown in Figure 1. In this example, a multiprocessor task with identifier "1" is described. The task is of type "computation" and has been executed on cluster "0" by eight processors $(0, 1, \ldots, 7)$.

2) Meta data: Additional to the basic information about tasks and clusters, Jedule also supports generic meta information. The meta information is later shown in the Jedule output for a better identification of the schedules. Meta information is simply defined by key/value pairs that characterize the current algorithm or the platform. The meta information helps the developer to later evaluate the impact of different parameters

```
<cmap name="standard_map">
    <conf name="min_font_size_label" value="11" />
    <conf name="font_size_label" value="13" />
    <conf name="font_size_axes" value="12" />

    <task id="computation">
        <color type="fg" rgb="FFFFFF" />
        <color type="bg" rgb="0000FF" />
    </task>

    <task id="transfer">
        <color type="fg" rgb="000000" />
        <color type="bg" rgb="f10000" />
    </task>

    <composite>
        <task id="computation" />
        <task id="transfer" />
        <color type="fg" rgb="FFFFFF" />
        <color type="bg" rgb="ff6200" />
    </composite>
</cmap>
```

Figure 2. Sample color map with one composite type.

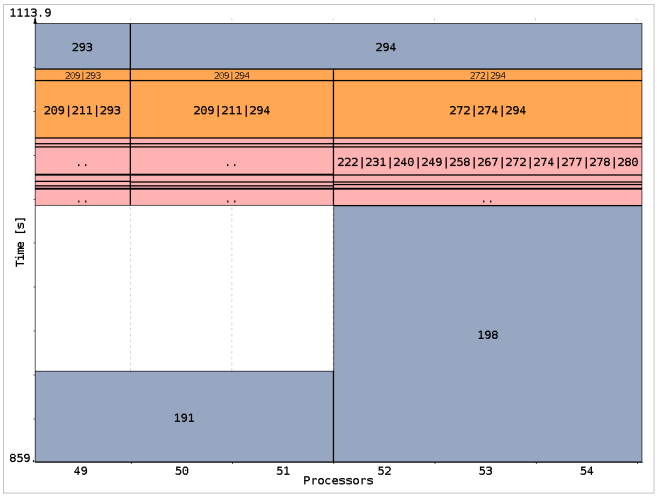

Figure 3. Example schedule featuring composite tasks (orange), which denote the overlapping of computation (blue) and communication time (red).

to an algorithm. A sample XML tag defining meta information for a scheduling algorithm is shown below.

```
<meta_info>
  <meta name="mindelta" value="-2"/>
  <meta name="maxdelta" value="2"/>
  <meta name="sort" value="comm"/>
</meta_info>
```

3) Composite tasks and time alignment: A parallel system may execute tasks concurrently on the same resources, i.e., a resource might be shared between tasks for an amount of time. Jedule supports overlapping tasks. For each resource which is shared by several tasks, Jedule creates a composite task. The identifier of a composite task is the concatenation of the single task IDs and the type is set to "composite". An example of the use of composite tasks is depicted in Figure 3. The schedule in this example contains two types of tasks, communication tasks, marked red, and computation tasks, marked blue. In order to mark the time when a host performs communication and computation operations at the same time, an orange composite task is introduced.

Another feature of Jedule is time alignment between clusters in a schedule. Each cluster schedule is a self-contained schedule, containing all tasks within this cluster. A schedule S^{C_j} for cluster C_j starts at time $t_s^{C_j}$ and ends at time $t_f^{C_j}$. The time $t_s^{C_j}$ ($t_f^{C_j}$) is defined as the minimal starting time (maximal finish time) of all tasks of cluster C_j. Each cluster may have totally different start and finish times. If all cluster schedules are displayed side by side, the developer is often interested in the overall utilization over all resources. Thus, Jedule supports two view modes: a scaled view and an aligned view. In the scaled view all clusters are displayed using their local minima and maxima of start and finish times. In the aligned view, the global minima and maxima of the task times are used to draw the schedules.

4) Color maps: One of the most important features of schedule visualizations is the coloring of tasks. Jedule supports user-defined color maps. The user can define a background and a foreground color for each task type. An example of a color map is presented in Figure 2. This color map defines the colors of the task types "computation" and "transfer" (data transfer). Moreover, the example also shows how the color of composite tasks can be specified. It would also be possible to add another composite task color scheme that contains a third type besides "computation" and "transfer". This method of coloring tasks gives the users enough flexibility to highlight the parts of the schedule that are important to them.

D. Using Jedule

Jedule supports two different modes to visualize schedules. The first mode is the interactive mode, which, when started, opens a window on the respective operating system and displays the schedule. The other is the command line mode, which is used to produce high quality graphics of schedules to be included in articles or reports.

1) Interactive mode: When Jedule is started in interactive mode, a Java Swing window is opened. The interactive mode is usually used when developing new scheduling algorithms. It supports several keyboard and mouse events that let the user investigate the details of schedules. The user can select which cluster, thus, which subset of the resources, should be displayed. Within each schedule graphic, the user can retrieve information about the tasks. Each task rectangle is labeled with the task identifier and painted with the predefined color. Additionally, the user can request detailed information for a task by clicking on the task. When information about a task is requested, Jedule displays the start and finish time of the task and the list of resources that were assigned to this task. This becomes very useful when tasks are overlapping or when multiprocessor tasks are scattered across resources and have no contiguous representation. The interactive mode also allows the user to zoom into the schedule or to move the current bounds of the schedule. To move the schedule (move the virtual window/frame above the schedule) the user can

drag the schedule with the mouse. The mouse wheel lets the user adapt the current view boundaries of the schedule (zoom in/out). The user can also zoom in by selecting a rectangular part of the current schedule. Jedule also supports fast rereads and redraws of the current schedule file using key strokes. This enables the developer of an algorithm to run a simulation and immediately see the produced schedule. In the interactive mode the user can also export the current view to one of the supported image formats or take a snapshot.

2) Command line mode: Jedule also has a command line interface. The command line interface is useful when the user wants an automated generation of a number of schedule graphics by using a script. Since Jedule is built upon the Swing toolkit, Jedule can easily support all file formats for which an export of the Swing graphics object exist. Currently, Jedule supports the graphic formats PNG, JPEG, and PDF. When a schedule is exported to a graphics format, the choice of the color map becomes essential as style guides of journals sometimes require gray scale graphics. Thus, the Jedule command line interface provides several parameters for adjusting the properties of the output graphic. The most important ones are the desired color map and the type of the output format. Besides that, the user can also specify the height and the width of the resulting graphic, or if the start and finish time of clusters should be aligned.

III. Case Study – Multiprocessor Task Scheduling

In the following sections, we introduce several use cases of Jedule. We show how Jedule has been used in these scenarios to improve an algorithm or to solve the investigated problem.

A. Introduction to M-Task Scheduling

In the present case study, we examine algorithms for the scheduling of mixed-parallel applications onto homogeneous clusters. A mixed-parallel application can be described as a directed acyclic graph (DAG) $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \{v_i \mid i = 1, \ldots, V\}$ is a set of vertices representing moldable tasks and $\mathcal{E} = \{e_{i,j} \mid (i,j) \in \{1, \ldots, V\} \times \{1, \ldots, V\}\}$ is a set of edges representing communication between tasks. A moldable task is a computational task that can be executed on varying numbers of processors. We denote by $T(v, p)$ the execution time of task v if it were to be executed on p processors (cores).

A homogeneous cluster H consists of h individual hosts, where each host has the same configuration (processors, memory, etc.). The hosts are interconnected by a network switch.

The objective of the scheduling algorithm is to find a mapping of the moldable tasks to the homogeneous cluster by minimizing the resulting makespan of the application. The scheduling algorithm has to determine the number of processors for each task, which increases the complexity of the problem. The algorithm also has to respect the computational order of the tasks, which is defined by the edges.

Several algorithms have been proposed that schedule mixed-parallel applications onto clusters [6], [7], [8], [9]. These algorithms reduce the completion time of the scheduled applications with regard to schedules that only exploit either task- or data-parallelism.

Over the last years we have worked on new algorithms as well as improving existing scheduling algorithms for mixed-parallel applications. Jedule has helped us understand the strengths and weaknesses of various algorithms.

B. Application of Jedule

One of the most recent algorithms for scheduling mixed-parallel algorithms onto homogeneous cluster is MCPA2 [10]. This algorithm is an extended version of the CPA algorithm (Critical Path and Area-based scheduling). The CPA algorithm attempts to find a good trade-off between the number of processors allocated to tasks and the length of the critical path. The critical path T_{CP} is longest path from the source node to the target node of a DAG, i.e., the sum of the execution times of the nodes along this path. CPA also relies on the metric T_A, which is defined as $T_A = \frac{1}{P} \sum_v (T(v, p(v)) \cdot p(v))$. The time (or area) T_A is a measure of how much a processor has to work on average. Both, T_{CP} and T_A are theoretical lower bounds of the makespan. CPA is a so-called two-step algorithm as it decouples the scheduling problem into two sub-problems. The first is the allocation phase, in which the algorithm determines the number of processors for each task. In the second step, the mapping phase, the algorithm tries to map the tasks with the precomputed allocation to the parallel platform. The decoupling of both steps usually decreases the computational complexity and thus, algorithms that implement this pattern are more likely to be used in practice. A main problem of CPA was addressed by Bansal *et al.* [7]. They showed that CPA often reduces the potential task parallelism of a DAG by letting allocations grow too big, as it does not consider the precedence levels of the graph. Bansal *et al.* proposed a new algorithm, called MCPA (modified CPA), which checks the total number of processors that are allocated to a precedence level. MCPA ensures that the number of processors allocated to a precedence level does not exceed the total number of processors in the system. It therefore favors task-parallelism for data-parallelism, which works well in many situations.

We compared the scheduling performance in terms of resulting makespan of CPA and MCPA in several scenarios. We conducted several thousand experiments with different types of DAGs (long, wide, serial, etc.) and multiple parallel platforms (from smaller cluster with 32 processors to bigger ones). The experiments were performed using a simulator, which was built on top of SimGrid [11]. We used Jedule to analyze the schedules obtained from the simulator. This allowed us to get a fast overview of the scheduling performance by viewing the scheduling output of CPA and MCPA side by side. By quickly browsing through the resulting schedules, we could isolate the case that is shown in Figure 4. The figure shows the visualization of the schedules produced by CPA (left) and MCPA. Both schedules have been created using the same DAG and parallel platform. However, one can observe that

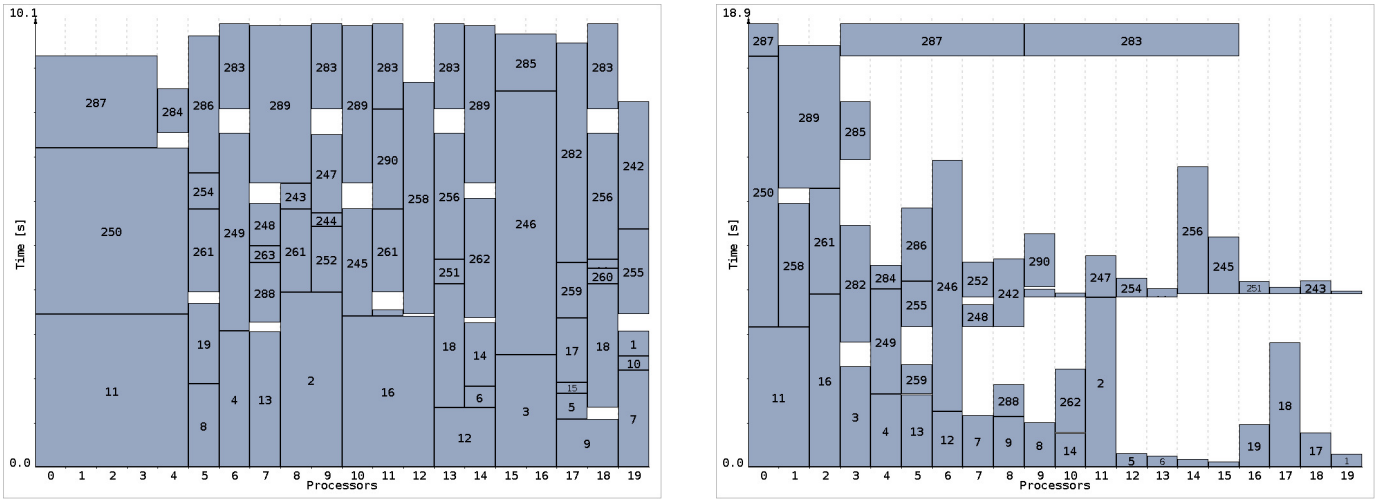

Figure 4. Jedule output for schedules produced by CPA (left) and MCPA (right). MCPA entails a load imbalance problem for this case.

the CPA algorithm exploits the computational resources of the cluster better than MCPA. In case of MCPA (on the right), the schedule contains large holes that correspond to idle CPU time. The main objective of any scheduler is to leave as few resources idle as possible. In this example, it can clearly be seen that the strategy of favoring task parallelism for data parallelism does not work. In the beginning, MCPA allocates one processor to each task in one precedence layer. However, MCPA restricts allocations from growing bigger as the number of processors of the corresponding precedence layer would exceed the total number of processors of the cluster. This strategy would still work well if each task in one layer had similar costs (operations to perform). But in the case considered, tasks in the precedence layer have different costs (e.g., tasks 2 and 5), which leads to a load imbalance. We could find a workaround to this problem by introducing a poly-algorithm (MCPA2) that uses CPA or MCPA depending on the DAG and the parallel platform. For the example shown in Figure 4 the poly-algorithm MCPA2 generates the same schedule as CPA.

In the case study presented, Jedule has helped us tremendously to quickly obtain an overview of different scheduling scenarios. We have used the PDF export function of Jedule to create documents with hundreds of schedule pictures.

IV. Case Study – Multi-DAG Scheduling on Homogeneous Clusters

A. Introduction

Here we extend the framework presented in the previous section to the case of scheduling several mixed-parallel application on homogeneous clusters. The definitions of a mixed-parallel application and a homogeneous cluster still hold in this section. The main change is that a batch of N distinct applications has to be scheduled.

In this scheduling problem two performance metrics have to be optimized simultaneously. The first metric measures the performance of the whole batch of mixed-parallel applications, i.e., the *overall makespan* defines the maximum completion time among the scheduled applications. The second metric quantifies the fairness of a schedule. A perfectly fair schedule is one in which all applications have the same stretch. The stretch of an application is defined as the makespan achieved in the presence of resource contention divided by the makespan that would have been achieved if the application had had dedicated use of the cluster. For instance, if a mixed-parallel application could have run in 2 hours using the entire cluster, but instead ran in 6 hours due to competition with other applications, then its stretch is 3. This is the most widely accepted definition in the literature, with a lower value denoting better performance.

Three approaches have been proposed in the literature that relate to the above problem. In the first approach multiple task graphs are combined into one and then a standard task graph scheduling heuristic is used. Algorithms following the second approach give a subset of the available processors to each application and schedule each of them on its subset using a known scheduling algorithm. The third approach relies on bi-criteria algorithms for scheduling independent moldable jobs, based on an approximation algorithm for optimizing the makespan. Enhanced algorithms derived from these approaches have been described and evaluated in [12].

B. Application of Jedule

Jedule was a great help for the evaluation conducted in [12]. Here we detail how it helped to check the validity of one of the proposed approaches to schedule multiple mixed-parallel applications on a single cluster. The approach proposed in [13] consists of distributing the processors of the cluster among the applications to schedule. Each application then has to build its own schedule according to this *constraint resource allocation* (CRA). The initial distribution of the processors can be done according to different characteristics of the submitted applications. For instance, in the `CRA_WORK` algorithm proposed in [13], each application i gets a share of resources β_i propor-

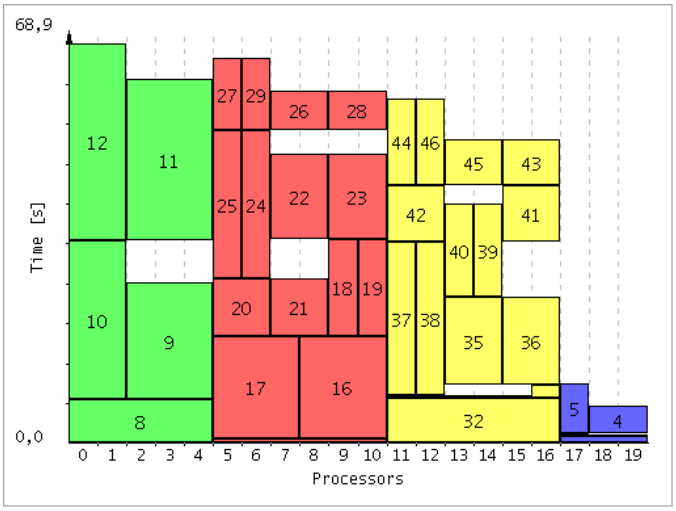

Figure 5. Jedule output for the schedule produced by the CRA_WIDTH algorithm. Four mixed-parallel applications, each having its own color, are scheduled on a cluster of 20 processors. The resource constraints imposed by the algorithm are respected.

tional to its own work. The work needed by a mixed-parallel application i is equal to $W(i) = \sum_{v_j \in \mathcal{V}_i} (T(v_j, p(v_j)) \cdot p(v_j))$ and β_i is formally defined as

$$\beta_i = \frac{\mu}{|\mathcal{A}|} + \frac{(1-\mu)W(i)}{\sum_{j \in \mathcal{A}} W(j)},$$

where $\mathcal{A}$ is the set of concurrent applications and μ can vary in $[0; 1]$ to give more importance to the work while distributing the resources.

A critical issue for such an algorithm is to ensure that each schedule respects its resource constraint. Thanks to the color map that assigns a different color to each application, it is easy to see the distribution of resources among the applications. Figure 5 shows a schedule produced by CRA_WORK algorithm for four mixed-parallel applications. We can see that the tasks of each application are mapped on distinct processors. The visualization offered by Jedule confirms that the algorithm does what it was designed for. It also points out that the initial distribution of the processors among the applications can be too restrictive. For instance, processors 17 to 19 are clearly underused. Such information which could be extracted from text logs, but with more efforts, immediately highlights the need for more complex algorithms.

In this context, Jedule was also used to see the impact of a conservative backfilling step applied at the end of the scheduling process. A comparison of the Jedule outputs with and without backfilling allows for a check that no task is delayed by this step. The reduction of the total idle time can also be easily quantified.

V. Case Study – DAG Scheduling on Heterogeneous Platforms

A. Introduction

In this third case study, we consider simpler applications, i.e., scientific workflows represented by task graphs made of single-processor tasks, and more complex execution platforms, i.e., a heterogeneous multi-cluster. More precisely, we select for this case study the scheduling of an instance of the Montage workflow [14], [15] with 50 compute nodes on a heterogeneous platform. Montage is a popular application in astronomy to create mosaics from distinct input images. The structure of the Montage workflow is given in Figure 6.

The target execution platform is depicted in Figure 7. For the sake of simplicity, this platform is composed of only four clusters. Two of them comprise four processors running at 1.65 Gflop/s (billions of operations per second), while the two other clusters only have two processors running twice as fast (3.3 Gflop/s). Each processor has its own communication link. Processors within a cluster are interconnected through a switch. Finally all clusters are interconnected by a single backbone. Note that such an infrastructure can easily be transformed into a set of bi- and quad-core processors connected on a LAN.

To schedule an instance of the Montage workflow on this platform, we selected the well-known Heterogeneous Earliest Finish Time (HEFT) algorithm [16]. The HEFT algorithm sorts the ready tasks of the application task graph by decreasing *upward rank*. Basically, the upward rank is the length of the critical path from a task to the exit task, including the computation cost of this task. It is the sum of the average execution cost of this task over all available processors and a maximum computed over all its successors. The terms of this maximum are the average communication cost of an edge and the upward rank of the successor. HEFT then uses the Earliest Finish Time (EFT) as the objective function for selecting the best processor for a node. The EFT of a task is the sum of its Earliest Start Time (EST) and its execution time on a candidate processor. The EST is the moment when the execution of a task can actually begin on a processor. An execution can start either when a processor becomes available or when all needed data has arrived on the processor.

B. Application of Jedule

Figure 8 shows the Jedule output of the schedule produced by HEFT for the considered Montage instance on the platform of Figure 7. This schedule was obtained in simulation.

In this figure we can see the multi-cluster view of Jedule mentioned in Section II. More interestingly, we can see that the last task executed on processor 2 implies a strange scheduling decision. This task is an mBackground task, according to the Montage workflow. The three other tasks of this kind are respectively executed on processors 9, 10, and 11.

This output tends to indicate a flaw in the scheduling algorithm. To confirm this graphical intuition allowed by Jedule, we checked the logs of the scheduling process. It appeared

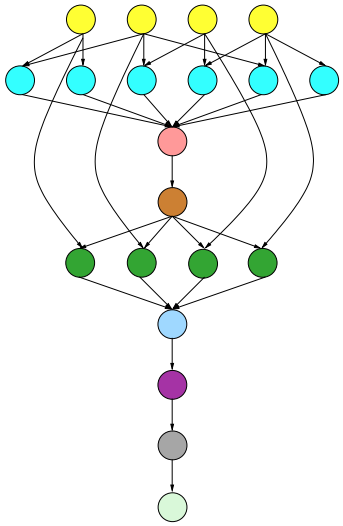

Figure 6. Structure of the Montage workflow (nodes with the same color are of same task type).

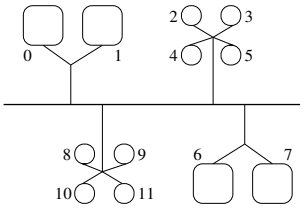

Figure 7. Heterogeneous platform used for the case study.

that processor 2 led to the earliest finish time for this task, and thus, the scheduling decision was correct. Nevertheless a problem exists. In presence of inter-task communications, as in Montage, moving a task from one cluster to another with exactly the same type of processors should lead to a greater finish time. This scheduling decision shows the opposite: sending data to another cluster is as costly as executing the task locally. The reason for the strange behavior, pointed out by Jedule, was in fact the description of the execution platform used for the simulation. The latency of the backbone connecting the different clusters was the same as the one for the links connecting the processors of a same cluster. In a reality the inter-cluster latency is usually much higher than the intra-cluster latency. We modified our description of the execution platform to obtain a more realistic setting. The schedule obtained on this modified platform is shown in Figure 9. We can see that this schedule does not exhibit odd scheduling decisions. The two fast clusters (processors 0-1 and 6-7) are chosen first and then the slower clusters are used. With regard to Figure 8, we can see that one of these slow clusters is more heavily used. This reflects the impact of the greater backbone latency on the scheduling decisions.

In this case study the overall makespan is the same for both schedules (140.9 seconds). If we had only relied on this metric to detect suspect behaviors, we would have missed the issue highlighted by Jedule.

VI. Case Study – Load Balancing on NUMA Architectures

A. Introduction

Dynamic load balancing of work units is often used to handle irregular computations. In case of parallel loop scheduling some iterations may require more computations than other iterations if, for example, the number of computations depends on input data or the amount of data sent or received differs. Using a static loop distribution may induce a load imbalance. Applications with recursive computations may use parallel tasks for each recursive function call. Often the number of calls and therefore the number of tasks is not known beforehand, and so a dynamic task scheduling is required to balance the parallel work across all processors. Identifying load imbalance in these applications is important in order to improve the execution scheme by changing the application or the scheduling algorithm.

In the present case study, we consider a task pool, which stores executable tasks in a virtually shared data structure accessible by all processors. Figure 10 shows an example of the task-based execution scheme. The actual storing may use central or distributed data structures for efficient access but these details are hidden behind the task pool interface. The task pool framework considered here especially targets fine-grained tasks. Hence, a low overhead of the task pool is an important requirement, which makes finding bottlenecks harder.

Similar approaches to exploit irregular parallelism are used, for example, in Intel's Threading Building Blocks (TBB) [4] for C++. Cilk [17] utilizes a fork-join-model to detach tasks for parallel execution. Task parallelism is also supported in OpenMP 3.0 [18].

B. Application of Jedule

The task pool run-time environment is able to log run-time information about each tasks for offline analysis in Jedule. The run-time environment stores for each thread the time used for executing a task and the time to get new tasks (or wait for new tasks if necessary). In Figure 10 the so-called *waiting time* covers the time for *get()* and *free()* calls while the so-called *task size* covers the time for *execution()*. Jedule can be used to visualize these run-time information to show more details about the actual utilization over time. In a case study we consider the parallel Quicksort, which creates two tasks for sorting each sub-array. At the beginning, there is only one task for the whole input array. After array partitioning there will be two new tasks, which will create another four tasks in total and so on. It is clear that due to the initial limited parallelism a linear speedup cannot be achieved. However, in theory, after $log(p)$ steps (with p being the number of total processors used for computation) every processor takes part in the sorting. Figure 11 shows the results for sorting 10 million random integers on an SGI Altix 4700 with 32 dual-core Itanium2 processors running at 1.6 GHz. Task execution times are highlighted in blue and waiting times are colored red. It can be noticed that due to an accidental bad choice

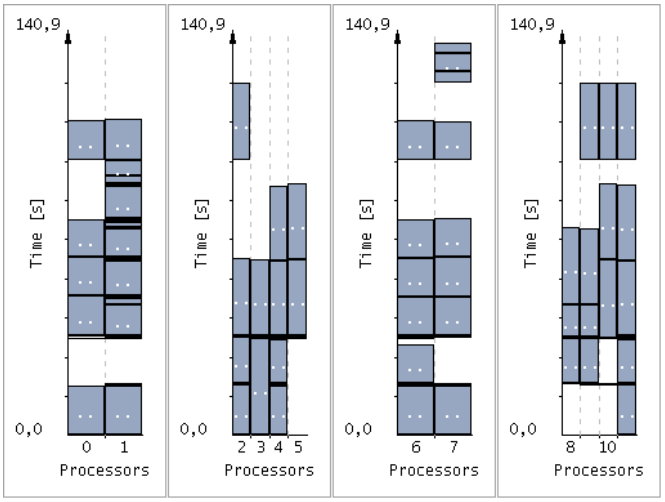

Figure 8. Jedule output of the schedule of a Montage instance on the heterogeneous platform described by Figure 7.

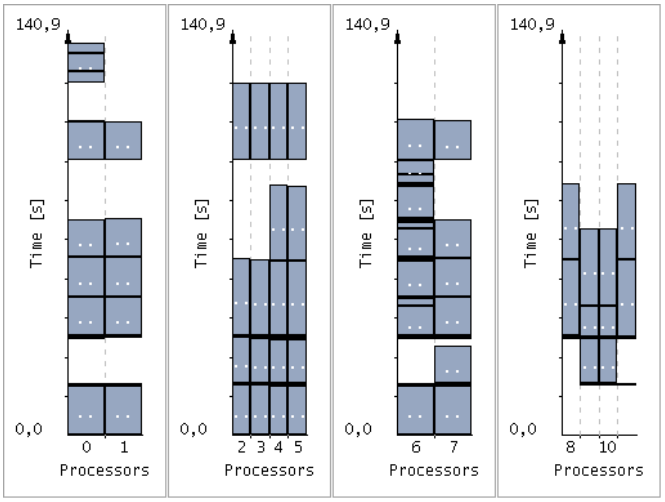

Figure 9. Jedule output of the schedule of a Montage instance on the heterogeneous platform with a greater latency on the backbone link.

```
struct Task { Function, Argument };

// initialization (master thread)
for ( each initial work unit U ) {
    TaskPool.create_initial_task( U.Function, U.Argument );
}

// working phase
parallel for ( each thread 1...p ) {
    forever() {
        Task T = TaskPool.get();
        if ( T == ∅ ) exit;
        T.execute(); // may create new tasks
        T.free();
    }
}
```

Figure 10. Task-based execution scheme.

of the pivot element, the initial array is not split into nearly equal-sized sub-arrays. In the actual situation, the small sub-array is split into other arrays creating small tasks for the other processors. The large sub-array takes more time to be partitioned into sub-arrays, so there is a long delay of the parallel execution. But even after a short period of parallel execution there are still some periods with low utilization with only 2-4 processors actually running.

With a specially crafted input array (inversely sorted numbers and selecting the middle element as pivot element) it is possible to force the Quicksort algorithm to equally partition the input array in each recursive step. One might expect a better utilization as after $log(p)$ steps enough tasks should be available for all processors. Figure 12 shows the actual utilization for 32 processors. In this case, only one processor is busy in almost half the total execution time. Since the processor has to swap every pair of numbers, it take much longer than for the random input case. After this initial tasks is finished two processors can start working concurrently, than 4 and so on. Interestingly, after some time of parallel execution with all processors, there is another hole where only a few processors are used. This is due to the high memory bandwidth requirements and the NUMA architecture of the machine. So, even two tasks with equal-sized arrays may take a different time to execute and therefore create new load imbalance.

The application of Jedule gives detailed information about the processor utilization of the system, which is harder to retrieve otherwise. The visualization helps to find unexpected waiting periods and gives an overview of the parallel execution and possible bottlenecks. Jedule can handle big data sets required to analyze fine-grained task parallel applications. In this case study, some experiments with the parallel Quicksort have created more than 200,000 individual tasks.

VII. Case Study – Parallel Workloads of Production Systems

Studying the workload of parallel systems is important to improve the job scheduler decisions and therefore to increase the throughput and efficiency of these systems. Several traces of parallel workloads are publicly available for scientific purposes. The Grid Workload Archive (GWA) and the Parallel Workload Archive (PWA) [19] contain several workload traces for different parallel architectures.

In a last case study, we use Jedule to obtain a bird's eye view of a parallel workload. Figure 13 shows the workload distribution for a 1024 node cluster (Thunder) at the Lawrence Livermore National Laboratory (LLNL). The graphic shows the workload of the cluster that was obtained on one day in 2007 (log file: `LLNL-Thunder-2007-0`, jobs: all jobs that finished on 02/02, log file source: PWA). On this day, 834 jobs were executed on that cluster. 20 nodes of this cluster were reserved as login and debug nodes, which can be seen in the graphic as jobs get only executed by nodes with a number greater than 20. We also highlighted in yellow the jobs of user 6447 to demonstrate how Jedule can support the analysis of parallel workloads of clusters or grids.

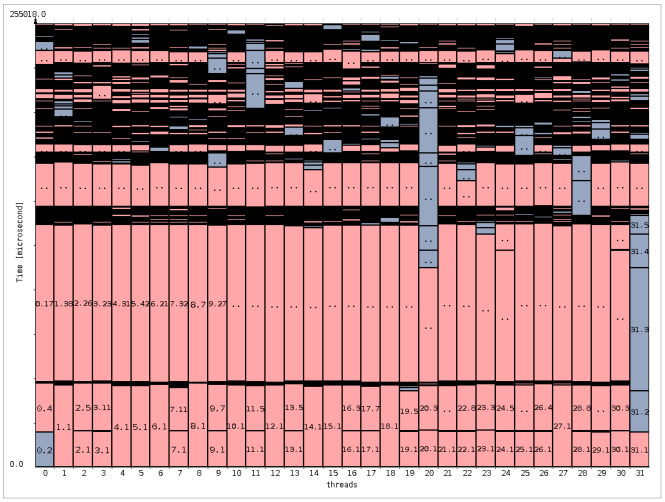

Figure 11. Quicksort with random 10,000,000 integers.

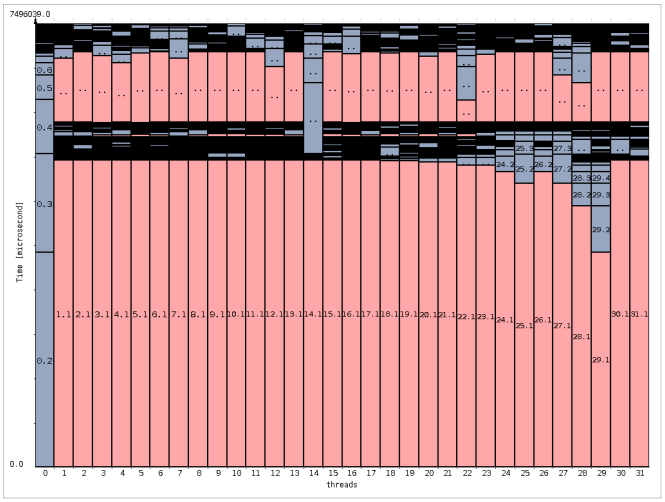

Figure 12. Quicksort with inversely sorted 200,000,000 integers.

VIII. Related Work

Visualization tools are often used to support the development of parallel programs. In many cases, the graphical representation of an executed application can help to identify bottlenecks and therefore scalability issues of the program.

A visualization tool that enables the developer to interactively investigate a trace of a parallel program is Pajé [20]. The tool is designed to support a potentially large number of communicating threads, primarily designed to tune a multithreaded molecular dynamic code. Pajé displays a program trace that has to be generated by running an instrument code. The tool supports many low level events like communication and synchronization between threads, using graphical elements like arrows between communicating threads. The Visual Trace Explorer (ViTE) is an extended version of Pajé, which also visualizes a sequence of events from a trace and additionally presents several statistics about the trace [21].

A tool that is also dedicated to understand scheduling algorithms is VizzScheduler [22]. It is part of a framework to develop and evaluate scheduling algorithms for the LogP cost model. So, one can improve the scheduling algorithms in a simulator before going to a real platform. VizzScheduler can be used to visualize program points during the actual execution of the program. It also visualizes scheduling algorithms by Gantt charts. The user can also alter several parameters of the LogP model via a graphical interface.

The GridSim-based Grid Scheduling Simulator Alea2 [23] also allows a graphical evaluation of the simulation. The tool provides graphical visualizations of several scheduling statistics, e.g., average system utilization, the number of running and waiting jobs, or the cluster usage.

A few other tools exist that help developers of parallel applications to analyze and tune their programs. Well known are VAMPIR [24] and TAU [25]. VAMPIR is usually used for visualizing MPI traces. The sequence of events is shown for each process using Gantt chart. It provides fine grained statistics of the program traced, e.g., PAPI counters. TAU is another tracing (and profiling) tool for HPC systems targeted to MPI applications. The generated traces can be graphically displayed in different viewers, e.g., Jumpshot [26] or VAMPIR.

For programs based on the PVM (Parallel Virtual Machine) the programming environment GRADE [27] provides graphical support. GRADE offers the developer a graphical program editor and contains a visualization tool to analyze the message-passing parallel programs.

A parallel program can also be visualized as a parallel execution graph [28]. The authors have shown how these execution graphs can be used to evaluate the performance of parallel programs that uses a distributed thread system (DTS). Graphs are generated from program traces and expose the structure of multithreaded programs.

IX. Conclusions

In this article we have introduced Jedule, a software tool that visualizes task schedules on parallel platforms. Jedule's main purpose is to provide an easy-to-use and generic tool for displaying arbitrary schedules as Gantt charts. It helps developers to get a first abstract overview (bird's eye view) of the decisions of the scheduling algorithms. Moreover, it can be used educationally to demonstrate how scheduling algorithms work. Jedule can be used to display the utilization of a parallel platform, e.g., a homogeneous cluster. Additionally, Jedule supports the grouping of resources into clusters. A cluster might be a real commodity cluster of PCs or just a single multicore machine, where each core is part of the bigger cluster. The schedule of each cluster can be viewed separately. The rectangles in the Jedule Gantt charts that represent user-defined events (a running job, a waiting time, or an I/O operation) can take a different color according to their type. Color maps can also be changed on the fly, thus, the user can highlight different events when investigating a schedule. Jedule provides two different modes to investigate schedules, the interactive mode and the command line mode. The command line mode helps users to produce high quality graphics of

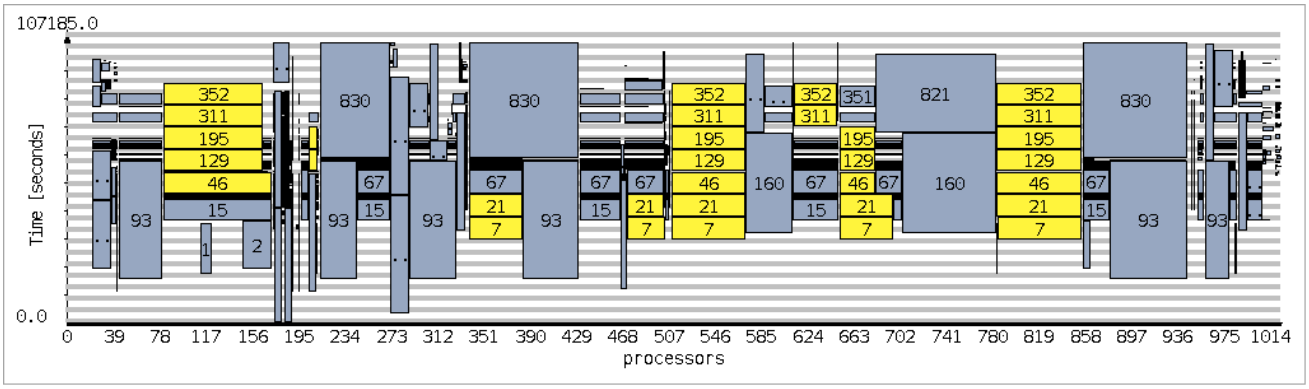

Figure 13. Visualization of the parallel workload of the LLNL Thunder Cluster on one day in 2007. Yellow rectangles denote jobs of a selected user.

schedules, which can later be embedded into documentations or articles. The interactive mode enables the user to focus on specific parts of the schedule by filtering or zooming into schedule events.

REFERENCES

[1] D. Thain, T. Tannenbaum, and M. Livny, "Distributed Computing in Practice: The Condor Experience," *Concurrency and Computation: Practice and Experience*, vol. 17, pp. 2–4, 2005.

[2] "OpenPBS website." [Online]. Available: http://www.openpbs.org/

[3] D. B. Jackson, Q. Snell, and M. J. Clement, "Core Algorithms of the Maui Scheduler," in *Revised Papers from the 7th International Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP '01)*. Springer, 2001, pp. 87–102.

[4] J. Reinders, *Intel threading building blocks*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2007.

[5] H. Brunst, D. Kranzlmüller, M. S. Muller, and W. E. Nagel, "Tools for scalable parallel program analysis: Vampir NG, MARMOT, and DeWiz," *Int. J. Comput. Sci. Eng.*, vol. 4, no. 3, pp. 149–161, 2009.

[6] N. Vydyanathan, S. Krishnamoorthy, G. Sabin, U. Catalyurek, T. Kurc, P. Sadayappan, and J. Saltz, "An Integrated Approach for Processor Allocation and Scheduling of Mixed-Parallel Applications," in *Proc. of the 35th Int. Conf. on Parallel Processing (ICPP'06)*, Aug. 2006, pp. 443–450.

[7] S. Bansal, P. Kumar, and K. Singh, "An Improved Two-Step Algorithm for Task and Data Parallel Scheduling in Distributed Memory Machines," *Parallel Computing*, vol. 32, no. 10, pp. 759–774, 2006.

[8] S. Hunold, T. Rauber, and F. Suter, "Redistribution Aware Two-Step Scheduling for Mixed-Parallel Applications," in *Proc. of the IEEE Int. Conf. on Cluster Computing (Cluster 2008)*, 2008, pp. 50 – 58.

[9] A. Radulescu and A. van Gemund, "A Low-Cost Approach towards Mixed Task and Data Parallel Scheduling," in *Proc. of the 15th Int. Conf. on Parallel Processing*, 2001.

[10] S. Hunold, "Low-Cost Tuning of Two-Step Algorithms for Scheduling Mixed-Parallel Applications onto Homogeneous Clusters," in *Proc. of the 10th IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Computing (CCGrid 2010)*, 2010.

[11] H. Casanova, A. Legrand, and M. Quinson, "SimGrid: a Generic Framework for Large-Scale Distributed Experiments," in *Proc. of the 10th IEEE Int. Conf. on Computer Modeling and Simulation*, Mar. 2008.

[12] H. Casanova, F. Desprez, and F. Suter, "On Cluster Resource Allocation for Multiple Parallel Task Graphs," Institut National de Recherche en Informatique et en Automatique, Research Report 7724, Mar. 2010. [Online]. Available: http://hal.archives-ouvertes.fr/inria-00461692/

[13] T. N'takpé and F. Suter, "Concurrent Scheduling of Parallel Task Graphs on Multi-Clusters Using Constrained Resource Allocations," in *Proceedings of the 10th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC)*, May 2009.

[14] "Montage Project," http://montage.ipac.caltech.edu/.

[15] A. Laity, N. Anagnostou, G. B. Berriman, J. C. Good, J. C. Jacob, D. S. Katz, and T. Prince, "Montage: An Astronomical Image Mosaic Service for the NVO," in *Proceedings of Astronomical Data Analysis Software and Systems (ADASS) XIV*, 2004.

[16] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 3, pp. 260–274, 2002.

[17] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou, "Cilk: An Efficient Multithreaded Runtime System," in *Proceedings of the 5th Symposium on Principles and Practice of Parallel Programming (PPOPP'1995*. ACM, 1995, pp. 55–69. [Online]. Available: citeseer.nj.nec.com/blumofe95cilk.html

[18] *OpenMP Application Program Interface, Version 3.0*, OpenMP Architecture Review Board, May 2008.

[19] D. Feitelson, "Parallel Workloads Archive (PWA)." [Online]. Available: http://www.cs.huji.ac.il/labs/parallel/workload/

[20] J. C. D. Kergommeaux, B. D. O. Stein, and M. S. Martin, "Paje: An Extensible Environment for Visualizing Multi-Threaded Program Executions," in *Proc. Euro-Par 2000, Springer-Verlag, LNCS*, 1900, pp. 133–144.

[21] "Vite (visual trace exlporer)." [Online]. Available: http://vite.gforge.inria.fr

[22] W. Löwe and A. Liebrich, "VizzScheduler - A Framework for the Visualization of Scheduling Algorithms," in *Euro-Par '01: Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing*. Springer, 2001, pp. 62–66.

[23] D. Klusáček and H. Rudová, "Alea 2 – Job Scheduling Simulator," in *Proceedings of the 3rd International Conference on Simulation Tools and Techniques (SIMUTools 2010)*, 2010.

[24] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach, "VAMPIR: Visualization and Analysis of MPI Resources," *Supercomputer*, vol. 12, pp. 69–80, 1996.

[25] S. S. Shende and A. D. Malony, "The Tau Parallel Performance System," *Int. J. High Perform. Comput. Appl.*, vol. 20, no. 2, pp. 287–311, 2006.

[26] O. Zaki, E. Lusk, W. Gropp, and D. Swider, "Toward Scalable Performance Visualization with Jumpshot," *Int. J. High Perform. Comput. Appl.*, vol. 13, no. 3, pp. 277–288, 1999.

[27] P. Kacsuk, J. Cunha, G. Dózsa, J. Lourenço, T. Fadgyas, and T. Antão, "A graphical development and debugging environment for parallel programs," *Parallel Computing*, vol. 22, no. 13, pp. 1747 – 1770, 1997, distributed and parellel systems: Environments and tools. [Online]. Available: http://www.sciencedirect.com/science/article/B6V12-3VV6MW4-P/2/26592872c685a5fb6d823063dab89a2a

[28] B. Steckelbach, T. Bubeck, U. Fößmeier, M. Kaufmann, M. Ritt, and W. Rosenstiel, "Visualization of parallel execution graphs," in *Graph Drawing*, ser. Lecture Notes in Computer Science, S. Whitesides, Ed., vol. 1547. Springer, 1998, pp. 403–412.